TU Clausthal

Clausthal University of Technology

A scalable runtime platform for multiagent-based simulation

Tobias Ahlbrecht, Jürgen Dix, Michael Köster, Philipp Kraus, Jörg P. Müller

IfI Technical Report Series IfI-14-02

Department of Informatics
Clausthal University of Technology

Impressum

Publisher: Institut für Informatik, Technische Universität Clausthal
Julius-Albert Str. 4, 38678 Clausthal-Zellerfeld, Germany
Editor of the series: Jürgen Dix
Technical editor: Federico Schlesinger
Contact: federico.schlesinger@tu-clausthal.de
URL: http://www.in.tu-clausthal.de/forschung/technical-reports/
ISSN: 1860-8477

The IfI Review Board

A scalable runtime platform for multiagent-based simulation

Tobias Ahlbrecht, Jürgen Dix, Michael Köster,
Philipp Kraus, Jörg P. Müller

Department of Informatics, Clausthal University of Technology
Julius-Albert-Str. 4
D-38678 Clausthal-Zellerfeld,Germany
firstname.lastname@tu-clausthal.de
http://www.in.tu-clausthal.de

Abstract

Using purely agent-based platforms for any kind of simulation requires to address the following challenges: (1) *scalability* (efficient scheduling of agent cycles is difficult), (2) *efficient memory management* (when and which data should be fetched, cached, or written to / from disk), and (3) *modelling* (no generally accepted meta-models exist: what are essential concepts, what implementation details?). While dedicated professional simulation tools usually provide rich domain libraries and advanced visualisation techniques, and support the simulation of large scenarios, they do not allow for "agentization" of single components. We are trying to bridge this gap by developing a *distributed, scalable runtime platform for multiagent simulation*, *MASeRaTi*, addressing the three problems mentioned above. It allows to plug-in both dedicated simulation tools (for the *macro view*) as well as the agentization of certain components of the system (to allow a *micro view*). If no agent-related features are used, its performance should be as close as possible to the legacy system used.

Contents

1 Introduction

In this paper, we describe ongoing work on a distributed runtime platform for multiagent simulation, *MASeRaTi*, that we are currently developing in a joint project[1]. The idea for *MASeRaTi* evolved out of two projects, Planets and MAPC.

Agent-based traffic modelling and simulation:
We developed *ATSim*, a simulation architecture that integrates the commercial traffic simulation framework *AIMSuN* with the multiagent programming system *JADE* (implemented in JAVA): *ATSim* was realized within Planets, a project on co-operative traffic management [2].

Agent-based simulation platform:
We implemented, in JAVA, an agent-based platform, *MASSim*, which allows several simulation scenarios to be plugged-in. Remotely running teams of agents can connect to it and *play against each other* on the chosen scenario. *MASSim* has been developed since 2006 and is used to realise the MAPC, an annual contest for multiagent systems.

While the former system centers around a commercial traffic simulation platform (*AIMSuN*), the latter platform is purely agent-based and had been developed from scratch. Such an agent-based approach allows for maximal freedom in the implementation of arbitrary properties, preferences, and capabilities of the entities. We call this the *micro-level*: each agent can behave differently and, possibly, interact with any other agent.

The traffic simulation platform *AIMSuN*, which works easily for tens of thousands of vehicles, however, does not suport such a micro-level view. Often we can only make assumptions about the *throughput* or other *macro*-features. Therefore, with *ATSim*, we aimed at a hybrid approach to traffic modelling and integrated the *JADE* agent platform in order to describe vehicles and vehicle-to-X (V2X) communication within a multiagent-based paradigm. One of the lessons learned during the project was that it is extremely difficult to *agentize*[3] certain entities (by, e.g. plugging in an agent platform) or to add agent-related features to *AIMSuN* in a scalable and natural way.

Before presenting the main idea in more details in Section 2, we point to related work (Section 1.1) and comment about the overall structure of this paper.

[1] `http://simzentrum.de/en/projects/desim`
[2] `http://www.tu-c.de/planets`
[3] To agentize means *to transform given legacy code into an agent so that it belongs to a particular multiagent system (MAS)*. This term was coined in [28]. In [27], Shoham used the term *agentification* for this.

1.1 Related work

In the last decade a multitude of simulation platforms for multiagent systems have been developed. We describe some of them with their main features and note why they are not the solution to our problem. *Shell for Simulated Agent Systems (SeSAm) [21]* is an IDE that supports visual programming and facilitates the simulation of multiagent models. *SeSAm*'s main focus is on education and not on scalability.

GALATEA [8] is a general simulation platform for multiagent systems developed in Java and based on the High Level Architecture [23]. *PlaSMA* [13] was designed specifically for the logistics domain and builds upon JADE. *AnyLogic*[4] is a commercial simulation platform written in Java that allows to model and execute discrete event, system dynamics and agent-based simulations, e.g. using the included graphical modelling language. *MATSim*[5] was developed for large-scale agent-based simulations in the traffic and transport area. It is open-source and implemented in Java. The open-source simulation platform *SUMO* [22] was designed to manage large-scale (city-sized) road networks. It is implemented in C++ and supports a microscopic view of the simulation while it is not especially agent-based. *Mason* [25] is a general and flexible multiagent toolkit developed for simulations in Java. It allows for dynamically combining models, visualizers, and other mid-run modifications. It is open-source and runs as a single process. *NetLogo*[29] is a cross-platform multiagent modelling environment that is based on Java and employs a dialect of the Logo language for modelling. It is intended to be easily usable while maintaining the capability for complex modelling.

TerraME[6]) is a simulation and modelling framework for a terrestrial system which is based on finite, hybrid, cellular automata or *situated agents*. We are using a similar architecture (Section 3), but we add some features for parallelisation and try to define a more flexible model and architecture structure.

Most frameworks with IDE support are not separable, so the architecture cannot be split up into a simulation part (e.g., on a High Performance Computing (HPC) cluster) and a visualisation/modeling part for the UI. Therefore an enhancement with HPC structure produces a new design of large parts of the system. Known systems like *Repast HPC*[7] use the parallelisation structure of the message passing interface MPI[8], but the scenario source code must be compiled into platform specific code. Hence, the process of developing a working simulation requires a lot of knowledge about the system specifics.

[4] http://www.anylogic.com/
[5] http://www.matsim.org/
[6] http://www.terrame.org/
[7] http://repast.sourceforge.net/
[8] http://www.mcs.anl.gov/research/projects/mpi/

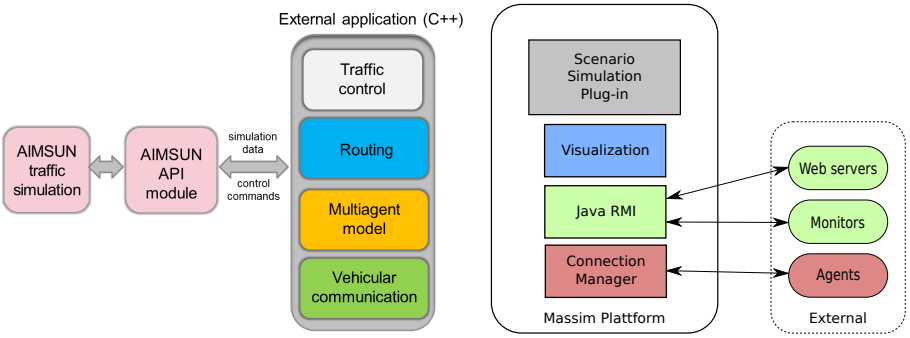
Repast HPC defines a parallel working agent simulation framework written in C++. In addition to our concept, *Repast* uses a similar structure to spawn environment and agents over the process and defines local and non-local agents. Technically, it uses Boost and Boost.MPI to create the communication between the processes. A dedicated scheduler defines the simulation cycle. A problem of *Repast HPC* is the "hard encoding" structure of the C++ classes, which requires good knowledge about the Repast interface structure. In our architecture, we separate the agent and scheduling structure into different parts, creating a better fit of the agent programming paradigm and the underlying scheduler algorithms.

Also, a number of meta models for multiagent-based simulation (MABS) have been developed so far. *AMASON* [20] represents a general meta-model that captures the basic structure and dynamics of a MABS model. It is an abstraction and does not provide an implementation. *MAIA* [14] takes a different approach by building the model on institutional concepts and analysis. The resulting meta-model is very detailed, focusing on social aspects of multiagent systems. *easyABMS* [12] provides an entire methodology to iteratively and visually develop models from which code for the *Repast Simphony* toolkit can be generated. The reference meta model for *easyABMS* is again very detailed making it possible to create models with minimal programming effort.

To summarize, we find that most platforms are either written in Java or are not scalable for other reasons. Many are only used in academia and simply not designed to run on a high performance computing (HPC) cluster. Common challenges relate to agent runtime representation and communication performance.

1.2 Structure of the paper

In Section 2 we discuss our past research (*ATSim* and *MASSim*), draw conclusions and show how it led to the new idea of a highly scalable runtime platform for simulation purposes. We also give a more detailed description of the main features of *MASeRaTi* and how they are to be realized. The main part of this paper is Section 3, where we describe in some detail our simulation platform, including the system meta-model and the platform architecture. Appendix 4 presents a small example on which we are testing our ideas and the scalability of the system as compared to *MASSim*, a purely agent-based approach implemented in Java. We conclude with Section 5 and give an outlook to the next steps to be taken.



(a) *ATSim* architecture (b) *MASSim* platform

Figure 1: Overview of the platforms

2 Essential features of *MASeRaTi*

In this section, we first present our own research in developing the platforms *ATSim* (Subsection 2.1) and *MASSim* (Subsection 2.2). We elaborate on lessons learned and show how this resulted in the new idea of the scalable runtime platform *MASeRaTi* (Subsection 2.3).

2.1 Traffic simulation (*ATSim*)

Most models for simulating today's traffic management policies and their effects are based on macroscopic physics-based paradigms, see e.g. [16]. These approaches are highly scalable and have proven their effectiveness in practice. However, they require the behaviour of traffic participants to be described in simple physical equations, which is not necessarily the case when considering urban traffic scenarios. Microscopic approaches have been successfully used for freeway traffic flow modelling and control [26], which is usually a simpler problem than urban traffic flow modelling and control, due to less dynamics and better predictability.

In [7], we presented the *ATSim* simulation architecture that integrates the commercial traffic simulation framework *AIMSuN* with the multiagent programming system *JADE*. *AIMSuN* is used to model and simulate traffic scenarios, whereas *JADE* is used to implement the informational and motivational states and the decisions of traffic participants (modelled as agents). Thus, all features of *AIMSuN* (e.g. rich GUI, tools for data collection and data analysis) are available in *ATSim*, while *ATSim* allows to simulate the overall behaviour of traffic, and traffic objects can be modelled as agents with goals, plans, and communication with others for local coordination and cooperation.

AIMSuN (Figure 1(a), left side) provides an API for external applications to access its traffic objects via Python or C/C++ programming languages. However, the *JADE*-based MAS (right side of Figure 1(a)) is implemented in Java. To enable *AIMSuN* and the MAS to work together in *ATSim*, we used CORBA as a middleware. Technically we implemented a CORBA service for the MAS and an external application using the *AIMSuN* API to access the traffic objects simulated by *AIMSuN*. The CORBA service allows our external application to interact with the MAS directly via object references. For details on the integration architecture, we refer to [7]. Two application scenarios were modelled and evaluated on top of *ATSim*: The simulation of decentralized adaptive routing strategies, where vehicle agents learn local routing models based on traffic information [11], and cooperative routing based on vehicle group formation and platooning [15]. The overall system shown in Figure 1(a) was developed in a larger research project and contained additional components for realistic simulation of V2X communication (extending the OMNET++ simulator), and for formulating and deploying traffic control policies; see [10].

Our evaluation of the *ATSim* platform using a mid-sized scenario (rush hour traffic in Southern Hanover, one hour, approx. 30.000 routes, see [10]) showed that while the agent-based modelling approach is intuitive and suitable, our integration approach runs into scalability issues. Immediate causes identified for this were the computationally expensive representation of agents as Java threads in Jade and the XML-based inter-process communication between Jade and the *AIMSuN* simulator. In addition, system development and debugging proved difficult because two sets of models and runtime platforms needed to be maintained and synchronised.

2.2 Multi-Agent Programming Contest (*MASSim*)

The *MASSim* platform [4, 3] is used as a simulation framework for the Multi-Agent Programming Contest (MAPC) [1][9]. Agents are running remotely on different machines and are communicating in XML with the server over TCP /IP. The server computes the statistics, generates visual output and provides interfaces for the simulation data while the simulation is running.

A drawback of dividing the simulation in such a way is the latency of the network that can cause serious delays. Network communication becomes a bottleneck when scaling up; the slowest computer in the network is determining the overall speed of the simulation. Running the simulation in one Java virtual machine leads to a centralised approach that might impede an optimal run (in terms of execution time) of a simulation.

Figure 1(b) depicts the basic components of the *MASSim* platform. *MASSim* will mainly serve us as a reference to compare scalability with *MASeRaTi*

[9] `http://multiagentcontest.org`

right from the beginning (using the available scenarios). We want to ensure that *MASeRaTi* outperforms *MASSim* in both computation time and number of agents.

2.3 *MASeRaTi*: The underlying idea

Our new simulation platform, *MASeRaTi*[10], aims at combining the versatility of an agent-based approach (the *micro-view*) with the efficiency and scalability of dedicated simulation platforms (the *macro-view*). We reconsider the three challenges mentioned in the abstract for using a purely agent-based approach.

Scalability: Efficient scheduling of agent cycles is a difficult problem. In agent platforms, usually each agent has his own thread. Using e.g. Java, these threads are realised in the underlying operating system which puts an upper limit of 5000 agents to the system. *These threads are handled by the internal scheduler and are therefore not real parallel processes.* In the *MASeRaTi* architecture we develop a micro-kernel where agents truly run in parallel. In this way, we reduce the overhead that comes with each thread significantly. We believe that this allows for a much better scalability than agent systems based on (any) programming language, where all processes are handled by the (black-box) operating system. Additionally, many simulation platforms use a verbose communication language (e.g., XML or FIPA-ACL) for the inter-agent communication that becomes a bottleneck when scaling up. We exploit the efficient synchronisation features of MPI instead.

Efficient memory management: Which data should when be fetched from disk (cached, written)? Most agent platforms are based on Java or similar interpreter languages. When using them we have no control over the prefetching or caching of data (agents need to access and reason about their belief state): this is done by the runtime mechanism of the language. We do not know in advance which available agent is active (random access), but we might be able to *learn* so during the simulation and thereby optimise the caching mechanism. This is the reason why we are using *Lua* in the way explained in the next section.

Modelling: As of now, no generally accepted meta-model for multiagent-based simulations exists. We would like to distinguish between essential concepts and implementation details. What are the agents in the simulation? Which agent features are important?

So the main problem we are tackling is the following: *How can we develop a scalable simulation environment, where the individual agents can be suitably pro-*

[10] `http://tu-c.de/maserati`

grammed and where one can abstract away from specific features? We would like to reason about the macro view (usually supported by dedicated simulation tools) as well as zooming into the micro view when needed. The overhead for supporting the microview should not challenge overall system scalability:

(1) If no agents are needed (no micro-view), the performance of *MASeRaTi* should be as close to the legacy code (professional simulation tools) as possible.

(2) If no legacy code at all is used, *MASeRaTi* should still perform better or at least comparable to most of the existing agent platforms (and it should have similar functionality).

Due to general considerations (Amdahl's law[17]) and the fact that not all processes will be parallelizable, it is not possible to achieve (1) perfectly (no agents: performance of *MASeRaTi* = performance of legacy code).

In addition to a scalable platform we also provide a *meta-model* for multi-agent-based simulations (MABS) and address the third challenge. However, the focus in this paper is on the first two challenges. The meta-model serves as a general starting point for the development of a MABS and ensures a certain structure of a simulation that is needed by the underlying platform in order to facilitate scalability. We have chosen *Lua* mainly because of its efficiency. It allows both object-orientation and functional programming styles and is implemented in native C. For details we refer to Section 3.2.

To conclude, we formulate the following basic requirements for *MASeRaTi*: (1) the support of a macro and micro view of a simulation, (2) a scalable and efficient infrastructure, and (3) a multiagent-based simulation modelling framework that also supports non-agent components.

3 Overview of the system

The overall architecture of our framework is inspired by concepts from *game developing*. The state of the art in developing massively multiplayer online role-playing games (MMORPG) consists in using a client-server architecture where the clients are synchronised during game play [9] via a messaging system. Well-known games include Blizzards's *World of Warcraft (WoW)* or EA's *SimCity 2013*, which supports multiplayer gaming with an "agent-based definition" in its own Glassbox engine[11].

While a game architecture is a good starting point for our purposes, we cannot create a server system with hundreds of nodes, which is powerful enough to handle a MMORPG system. For developing purposes we also need a single node-based system, which can run on a small (desktop) node. After

[11] `http://andrewwillmott.com/talks/inside-glassbox`

the developing process the source codes must then be transferable to a HPC system.

Our underlying meta-model uses the well established concept of a BDI-agent[27, 30] in a variant inspired by the agent programming language Jason [6] combined with the idea of an entity [2] that evolved out of the experiences gathered in the MAPC. Our agent model connects agents to these `entities` in the simulation world. Agents consist of a *body* and a *mind*: While the mind (being responsible for the deliberation cycle, the mental state etc.) does not have to be physically grounded, the entity has to be located in an area of the simulation. Thus, an entity is an object with attributes that an agent can control and that might be influenced by the actions of other agents or the overall simulation. Intuitively, an agent can be viewed as a puppet master that directs one (or more) entities. For all other objects in the simulation world, we use the concept of *artifacts* [5]. We also provide a basic notion of a computational `norm` that can be used by the simulation designer to steer the agents' behaviour. Additionally, all objects can be grouped by using `ObjectGroups`. See Section 3.3 for details.

3.1 Architecture

Our system is composed of three layers (Fig. 2):

Micro-kernel (MK): The micro-kernel is a C++ based system, which defines the basic network parallelisation scheduling algorithms. The layer defines the underlying structure, e.g. plug-in and serialization interface, Prolog interface for the belief base and statistic accumulation interface. The layer describes a meta-model for a parallel simulation (Section 3.2).

Agent-model layer (AML): The agent-model layer (Section 3.3) defines the model of an agent-based simulation and is written in Lua[12] [19]. Within this layer the relation and entities of an agent-based simulation are created e.g. BDI-agent, world, artifacts, etc. Due to the multiple-paradigm definition of Lua pure object-oriented concepts are not supported directly. Technically speaking, Lua uses only simple data types and (meta-) tables. Fortunately, based on these concepts, we can create an object-oriented structure in Lua itself. This allows us to work in a uniform fashion with UML models at the AML and the scenario layer.

Scenario layer (SL): The third layer is the instantiation of the AML with a concrete scenario, e.g., a traffic setting or the MAPC cow scenario. It is represented by dotted boxes in Fig. 2 to emphasize the difference to the AML layer. Section 4 provides an example.

[12] `http://www.lua.org/`

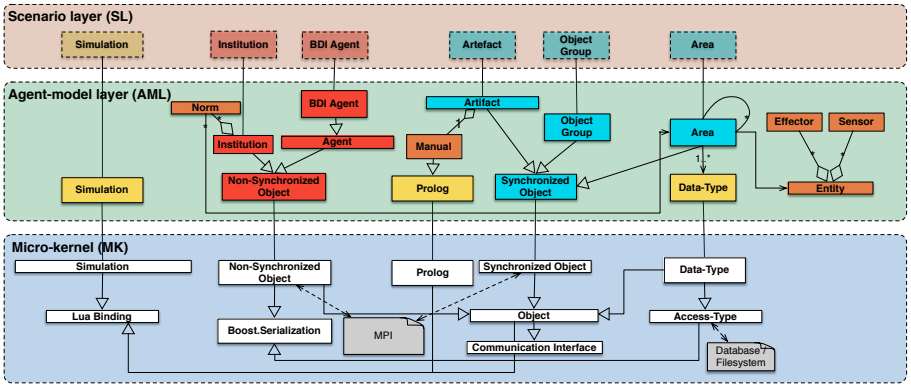


Figure 2: *MASeRaTi* system architecture: UML class diagram

An important aspect is the linkage between the three layers, and in particular the connections between the micro-kernel and the AML (illustrated in Fig. 2) and discussed further in the following sections.

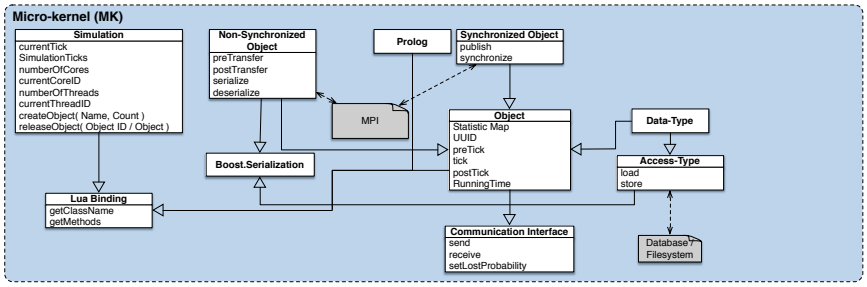
3.2 Micro-kernel

The micro-kernel describes the technical side of the system and is split up into two main structures (Fig. 3(b)). The core part (below) defines the scheduler algorithms, the core and memory management, the network and operating system layers and the plug-in API within a Prolog interpreter. Above these core utilities the Lua interpreter (top) is defined and each class structure on the core can be bound to "Lua objects". The Lua runtime is instantiated for each process once, so there is no elaborated bootstrapping.

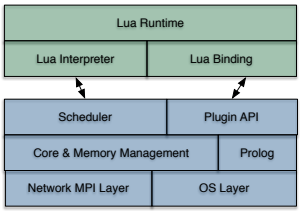
The choice of Lua is affected by the scaling structure and the game developing viewpoint. Lua, a multi paradigm language, has been used for game development for many years ([24]). An advantage of Lua is the small size of its interpreter (around 100 kBytes) and the implementation in native C with the enhancement to append its own data structures into the runtime interpreter with the binding frameworks. The multiparadigm definition of Lua, especially object-oriented and functional [19], can help us to create a flexible metamodel for our simulation model. Lua can also be used with a just-in-time compiler.

The kernel defines basic data structures and algorithms (Fig. 3(a)):

Simulation: A global singleton simulation object, which stores all global operations in the simulation e.g. creating agents or artifacts. It defines



(a) Micro-kernel: UML class diagram



(b) Architecture

Figure 3: Micro-kernel data model (a) and architecture (b)

the initialization of each simulation; the constructor of the Simulation object must create the world object, agent objects, etc.

Object: Defines the basic structure of each object within the simulation. All objects have got a UUID (Universally Unique Identifier), a statistical map for evaluating statistical object data, the (pre/post)tick methods to run the object and the running time data, which counts the CPU cycles during computation (for optimisation).

Prolog: An interface for using Prolog calls within the simulation.

Each class is derived from the *Lua Binding* class, so the objects will be mapped into the AML.

The mapping between the micro-kernel and the AML is defined using a *language binding concept*. The Lua interpreter is written in native C. Based on this structure, a C function can be "pushed" into the Lua runtime. The function will be stored into a global Lua table; the underlying C function is used with a script function call.

Our concept defines the micro-kernel in UML; instantiated C++ objects are mapped into the runtime environment by a Lua binding framework (e.g.

Lua Bridge[13] or Luabind[14]). Classes and objects in Lua are not completely separate things, as a class is a table with anonymous functions and properties. If a Lua script creates an object, it calls the constructur, which is defined by a meta-table function,the underlying C++ object will be also created and allocated on the heap. The destructor call to an object deterministically removes the Lua object and its corresponding C++ object. All C++ objects are heap allocated and encapsulated by a "smart pointer", as this avoids memory leaks. This concept allows consistent binding between the different programming languages and the layer architecture.

Each *Object* comes from the *Communication interface*, which allows an object to send any structured data to another object. The central *Object* inherits to three subclasses. This structures necessary for creating a distributed and scalable platform with optimisation possibility:

Synchronised Object:
An object of this type is synchronised over all instances of the micro-kernel (thread and core synchronised). It exists also over all instances and needs a blocking communication. In the agent programming paradigm the world must be synchronised.

Non-Synchronised Object:
This object exists only on one instance of the micro-kernel and can be transferred between different instances of the micro kernel. It should be used for agents and norms, because the evaluation is independent from other objects. Using the "execution time" of the tick (time complexity), we can group such objects together.

Data-Type:
This object represents a data structure, e.g. a multigraph for the traffic scenario with routing algorithms (Dijkstra , $A^\star$ and $D^\star$). The data types will be pushed into the micro-kernel with the plug-in API. The *Access-Type* creates the connection to the storing devices.

Synchronised and *non-synchronized* objects are implemented via Boost.MPI[15] structure, and the *Access-Type* defines the interface to a database or the filesystem for storing / loading object data. The access via the data interface will be defined by the Boost.Serialization library[15], so we can use a generic interface. Based on the *Data-Type* we can use the defined plug-in API for math datatypes (Fig. 4), which allows to create a (multi-) graph interface for our traffic scenario, based on Boost-Graph[15]. This enables us to use a differential equation solver like OdeInt[16] to simulate the macroscopic view in the

[13] `https://github.com/vinniefalco/LuaBridge`
[14] `http://www.rasterbar.com/products/luabind.html`
[15] `http://boost.org/doc/libs/release/libs/`
[16] `http://www.odeint.com/`

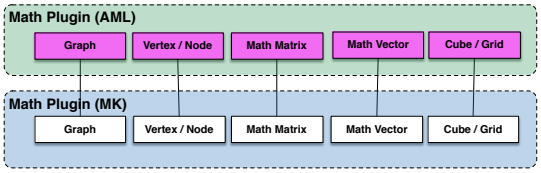


Figure 4: Math plug-in

simulation (e.g. a highway traffic model can be simulated with a differential equation while employing a microscopic agent-based view for an urban traffic area. The "glue" between these two types can be defined by a "sink / source data-type". A plug-in is defined in a two-layer structure. The plug-in is written in C++ and based on the Lua binding structure mapped into the higher layers. The plug-in interface is based on a native C implementation to avoid problems with name managing in the compiler and linker definition. Plug-ins are stored in a dynamic link library; they are loaded upon start of the kernel.

3.3 Agent-model Layer

The agent-model layer (AML) (depicted in 5) defines a meta-model of an agent-based simulation. It provides the basic structure and serves as a starting point for an implementation. We start by explaining the structure, followed by the overall system behaviour; we end with a general description of the development process. Realization details (pseudo code) can be found in the appendix (Section A.2).

3.3.1 Structure

The structure of the meta-model is heavily influenced by the goal of creating a simulation which can be distributed over several nodes or cores. In such a multiagent simulation, the developer has to decide for each object whether it has to be present on every single core or whether it can exist independent of the other objects (we aim for the latter). These two options lead to two approaches: (1) the invocation of functions in the same simulation step (the objects being on the same core), or, (2) the sending of messages (objects are not on the same core) after a simulation step. Since the first approach has the drawback to be forced to synchronise all objects, we choose the latter.

The goal of the AML is to simplify the development of multiagent simulations by defining those objects that have to be synchronised and those that run independently. A developer can easily modify the AML to her needs, in particular to redefine the synchronicity of objects.

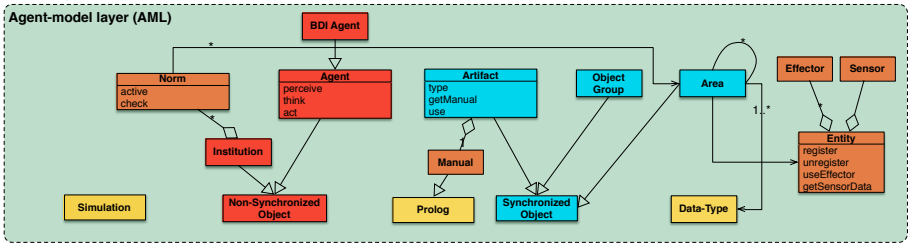


Figure 5: Agent-model layer: UML class diagram

Figure 5 illustrates the structure of the AML. Mainly, a simulation consists of a singleton `Simulation`, the non-synchronised object types `Agent`, `Norm`, and the synchronised classes `Area`, `Artifact`, `ObjectGroup`. While for the `Simulation` only one instance is allowed, the other objects can be instantiated several times. All instantiated objects are being executed in a step based fashion and therefore implement a `tick` method.

Simulation: The `simulation` class in the AML is the Lua-based counterpart to the `simulation` class in the MK. It is responsible for the creation, initialisation and deletion of objects, thus it is in full control over the simulation.

Agent: As we aim to simulate as many agents as possible we have to ensure that this part of the model can run independent of the rest. Therefore we define two kinds of agents as non-synchronised objects: a generic agent based on [30] and a more sophisticated BDI agent [27] inspired by Jason [6]. The agent interacts with the environment through `entities` [2]. In general an agent can have random access to the simulation world, so we can only encapsulate some parts of the agent, namely the internal actions and functions while the effects on the environment have to be synchronised. That is the reason for separating the agent into two parts: the mind (the `agent`) and the body (the `entity`). Thus, the generic agent has three methods that are invoked in that order: (1) `perceive`, (2) `think`, and (3) `act`. Inside these methods, we can call the methods of the entity directly while communication between objects has to be realised over a synchronised object (for instance with the means of an `artifact`). The agent developer has to explicitly specify the variables that have to be synchronised.

BDI Agent: The BDI agent is more sophisticated and consists of a `Belief Base` representing the current world view, a set of `Events` describing changes in the mental state, a set of plans `Plans`, and a set of `Intentions`

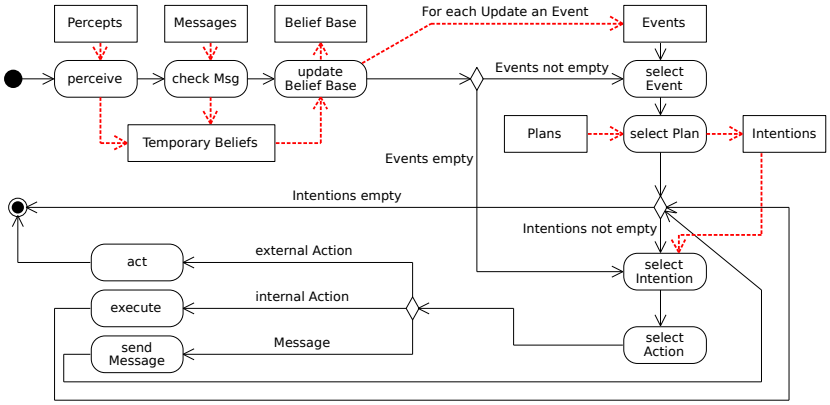


Figure 6: BDI agent cycle: Activity diagram and data flow

describing the currently executed plans. Fig. 6 shows an overview of the agent cycle. Black (continuous) lines represent the activity flow while red (dashed) lines show the data flow. The agent cycle is executed within the tick method. For each `tick`, the agent first perceives the environment, and checks for new messages. Based on this information, the belief base gets updated and an event for each update is generated. From the set of events one particular event is selected and a plan that matches this event will be chosen and instantiated. During a simulation run this might result in multiple instantiated plans at the same time and allows the agent to pursue more than one goal in parallel. Being a BDI agent it can only execute one action at a time, but several internal actions per simulation `tick`. The next method selects the next action of an instantiated plan (i.e. the next action of an intention). In contrast to Jason, the agent cycle does not stop here if it was an internal action or a message, i.e., an action that does not affect the environment. Thus, the agent selects the next event (if possible) or next intention (if possible) until it reaches a global timeout (set by the simulation) or an external action is executed that forces a synchronisation, or if the set of events and intentions are both empty. Again, the agent developer has to explicitly tell the simulation platform the variables that have to be synchronised.

Artifact: For all passive objects of a simulation we use the `artifact` methodology defined in [5]. Basically, each artifact has a `type` and a `manual` in Prolog (a description of the possible actions associated with it) and a `use` method that allows an agent to execute a particular action. Due to the generality of this approach the developer decides whether the actions are known by the agents beforehand or not. Additionally, since

the artifact is defined as a synchronous object, one can consider a derivation of this object that implements the actions as methods and allows for direct method invocation.

Area: So far, we defined the main actors of a simulation but how are they connected among each other? An artifact does not have to be located inside a real simulation, i.e., it does not need a physical position (in contrast, most objects do need one). Therefore, we define an `area` as a logical or physical space (similar to the term *locality* introduced by [18]). There can be several areas, subareas, and overlapping areas. In the general case, agents have random access to the environment, so the areas have to be synchronised over all cores of the simulation platform. In some circumstances, however, it is reasonable to create a new class inheriting all properties from the non-synchronised object. Within an area, we define some basic data structures and algorithms for path finding, etc. The most important issue, the connection of the non-synchronised agents with the synchronised areas is realised by the use of entities. Agents perceive the environment and execute actions by using the entities' sensors and effectors.

Entity: An entity can be seen as the physical body of an agent located inside an area. An agent can register to it, get the sensor data, and execute actions that possibly change the environment. The entity has some effectors and sensors that are easily replaceable by the simulation developer. Since such an entity represents the physical body of an agent and is meant to connect an agent with the environment it has to be synchronised over all cores.

Institution & Norm: An institution is an object that checks for norm violations and compliance. More precisely, it operates as a monitor and is also responsible for sanctioning. But a developer can also decide to separate these two tasks. For the future, we are planning to focus only on three kinds of norms: obligations, permissions, and prohibitions. Additionally, we will only consider exogenous norms (events that occur in at least one area) and not rules that affect the agent's mind, plans etc. Due to the non-synchronisation, the agent developer has to tell the simulation platform the variables that have to be synchronised.

ObjectGroup: Finally, an `ObjectGroup` – as the name implies – defines a group of objects. It can be used to group agents, artifacts or other objects. Method calls on an `ObjectGroup` are forwarded to all group members, i.e., with a single method call, all corresponding methods (with the same type signature) of the group members are invoked. In order to reduce overhead and to avoid circular dependencies we only allow a flat list of members at the moment. However, if a hierarchy is needed, it can be easily implemented.

3.3.2 Agent-model layer behaviour

So how does the overall behaviour look like? Initially the simulation object creates a number of agents, areas, object groups, norms, etc., and changes the global properties in the three phases: `preTick`, `tick`, and `postTick`. It can delete and create new agents during runtime. However, if the simulation developer decides to allow an agent to create another agent, this is consistent with the meta-model. The agent cycles are executed in each `tick` method, also the artifacts', norms' and areas' main procedures are executed in this phase. The `preTick` is most often used as a preparation phase and the `postTick` phase is used for cleaning up.

This section contains some heavy technical machinery and describes even some low level features that are usually not mentioned. However, our main aim is to ensure scalability in an agent-based simulation system. In order to achieve that, we came up with some ideas (using Lua and how to combine it with BDI-like agents) that can only be understood and appreciated on the technical level that we have introduced in this section.

4 Evaluation: Cow scenario

Scalability is an important aim of the platform and therefore has to be evaluated early on. For that reason we chose the *cow scenario* from the MAPC as a first simulation that is realistic enough in the sense that it enforces the cooperation and coordination of agents. As it is already implemented for the *MASSim* platform, it can easily serve as a first benchmark.

In addition, we can test the suitability of the proposed meta-model and test a first implementation. Furthermore, the cow scenario contains already some elements of more complex scenarios like the traffic simulation.

The cow scenario was used in MAPC from 2008 to 2010. The task for the agents is to herd cows to a corral. The simulated environment contains two corrals – one for each team – which serve as locations where cows should be directed to. It also contains fences that can be opened using switches. Agents only have a local view of their environment and can therefore only perceive the contents of the cells in a fixed vicinity around them. A screenshot of the visualisation as well as a short description of the scenario properties are depicted in Fig. 7. For a detailed description we refer to [3]. Using the proposed meta-model AML we can now implement the cow scenario in the following way[17].

Fig. 8 shows how we derived the cow scenario classes from appropriate superclasses of the agent-model layer. The grid of the environment is implemented as an `Area`. Obstacles are defined by a matrix that blocks certain

[17] Please note, that this is ongoing work. The corresponding Lua code can be found in the appendix, Section A.2.

cells. The two corrals are subareas located inside the main area. Fences will become `Artifacts`. Similarly, we define a switch as an artifact that controls and changes the state (opened or closed) of a fence when getting activated. The cows are realised by a reactive agent that perceives the local environment and reacts upon it. For such a reactive agent the basic `Agent` definition together with an `entity` representing the cow are sufficient, while for the cowboy agents we need a more complex behaviour that facilitates coordination and cooperation. For this reason we use the `BDIAgent` (recall Fig. 6) class and create an `entity` for each cowboy agent. Furthermore, for each entity we create a simple `MoveEffector` that can be used by the entities to alter their position and a `ProximitySensor` providing the entities with their percepts. Additionally, we have to define the two teams by using the notion of an `ObjectGroup`. Finally, the `simulation` creates all agents and entities, assigns them to the two teams and creates the simulation world.

To conclude, this preliminary evaluation shows that it is possible to express each aspect of the scenario using the predefined classes without the need to derive further ones from the synchronised or non-synchronised objects. (Nonetheless, doing so still remains a possibility). Regarding the suitability of Lua, it is an extremely flexible language that comes at the cost of a certain degree of usability: any newcomer needs some time to master it. But even then, having appropriate tools and methodologies that support the modelling process is a necessity to ensure an improved workflow and reduced error-proneness.

5 Conclusion and outlook

In this paper, we described ongoing work towards a distributed runtime platform for multiagent simulation. The main contributions of this paper are: (1) an analysis of the state of the art in agent-based simulation platforms, leading to a set of requirements to be imposed on a simulation platform, focusing on runtime scalability and efficient memory management; (2) the proposal of a novel architecture and design of the *MASeRaTi* simulation platform, bringing together a robust and highly efficient agent kernel (written in Lua) with a BDI agent interpreter including multiagent concepts such as communication and computational norms; and (3) an initial proof of concept realization featuring a simple application scenario.

The work presented in this paper provides the baseline for further research during which the *MASeRaTi* system will be extended and improved. Issues such as optimisation of the scheduler and the caching mechanisms sketched in the appendix (Section A.2) will be explored in more detail. Also, systematic experimental evaluation will be carried out using more sophisticated and much larger traffic simulation scenarios. As the *ATSim* platform introduced in Section 2.1 can deal with a few thousand (vehicle) agents, we ex-

pect *MASeRaTi* to scale up to one million agents. By the time we prepare the version of this paper for the postproceedings, we shall have more information available with respect to evaluation methods, criteria, and metrics, including but not restricted to scalability. Aside, different communication technologies like Bittorrent[18]) for the inter-object communication will be investigated.

Given the three objectives in the abstract, our focus in this paper has been on the first two: scalability and efficient memory management, whereas we only touched the third, modelling. Here, one avenue of research is to develop appropriate modelling tools to support the *MASeRaTi* architecture. Finally, methodologies for simulation development will be explored, starting from established methodologies such as GAIA, Tropos, or ASPECS.

[18] `http://www.libtorrent.org/`

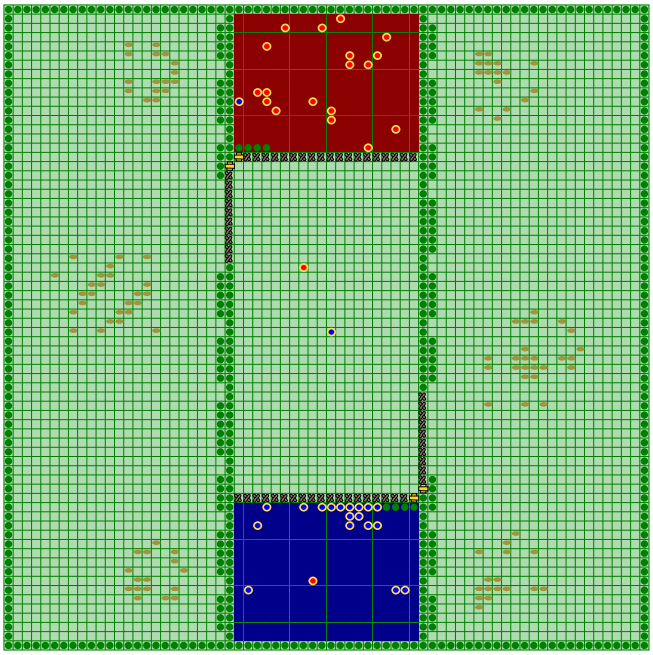


Figure 7: The environment is a grid-like world. Agents (red (at top) and blue (at the bottom) circles) are steered by the participants and can move from one cell to an adjacent cell. Obstacles (green circles) block cells. Cows (brown circles) are steered by a flocking algorithm. Cows tend to form herds on free areas, keeping the distance to obstacles. If an agent approaches, cows get frightened and flee. Fences (x-shapes) can be opened by letting an agent stand on a reachable cell adjacent to the button (yellow rectangles). An agent cannot open a fence and then definitely go through it. Instead it needs help from an ally. Cows have to be pushed into the corrals (red and blue rectangles).

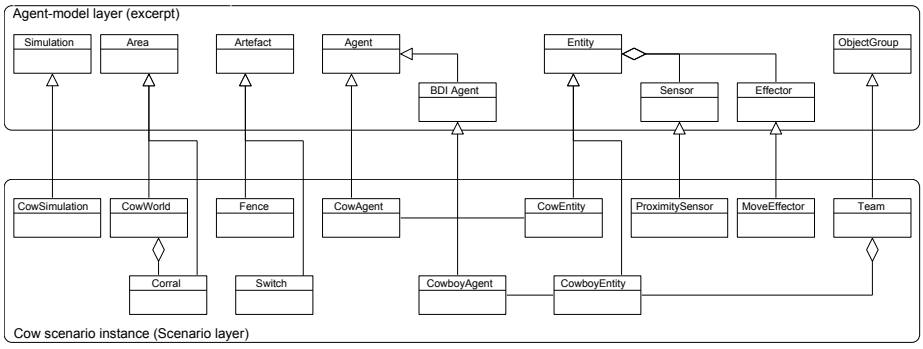


Figure 8: Cow scenario: UML class diagram

References

[1] Tobias Ahlbrecht, Jürgen Dix, Michael Köster, and Federico Schlesinger. Multi-Agent Programming Contest 2013. In Massimo Cossentino, Amal Fallah Seghrouchni, and Michael Winikoff, editors, *Engineering Multi-Agent Systems*, volume 8245 of *Lecture Notes in Computer Science*, pages 292–318. Springer Berlin Heidelberg, 2013.

[2] Tristan Behrens. *Towards Building Blocks for Agent-Oriented Programming*. PhD thesis, Clausthal University of Technology, 2012.

[3] Tristan Behrens, Mehdi Dastani, Jürgen Dix, Michael Köster, and Peter Novák. The Multi-Agent Programming Contest from 2005-2010. *Annals of Mathematics and Artificial Intelligence*, 59:277–311, 2010.

[4] Tristan M. Behrens, Jürgen Dix, Mehdi Dastani, Michael Köster, and Peter Novák. Technical Foundations of the Agent Contest 2008. Technical Report IfI-08-05, Clausthal University of Technology, December 2008.

[5] Rafael H. Bordini, Mehdi Dastani, Jürgen Dix, and Amal El Fallah-Seghrouchni, editors. *Multi-Agent Programming: Languages, Tools and Applications*. Springer, Berlin, 2009.

[6] Rafael H. Bordini, Jomi F. Hübner, and Michael Wooldridge. *Programming multi-agent systems in AgentSpeak using Jason*. Wiley & Sons, 2007.

[7] Viet-Hung Chu, Jana Görmer, and Jörg P. Müller. ATSim: Combining AIMSUN and Jade for agent-based traffic simulation. In *Proceedings of the 14th Conference of the Spanish Association for Artificial Intelligence (CAEPIA)*, volume 1. AEPIA, 2011. Electronic Proceedings.

[8] Jacinto Dávila and Mayerlin Uzcátegui. Galatea: A multi-agent simulation platform. In *Proceedings of the International Conference on Modeling, Simulation and Neural Networks*, 2000.

[9] Mattijs Driel, Jos Kraaijeveld, Zhi Kang Shao, and Remco van der Zon. A Survey on MMOG System Architectures, 2011.

[10] Maksims Fiosins, Jelena Fiosina, Jörg P. Müller, and Jana Görmer. Reconciling Strategic and Tactical Decision Making in Agent-oriented Simulation of Vehicles in Urban Traffic. In *Proceedings of the 4th International ICST Conference on Simulation Tools and Techniques*, SIMUTools '11, pages 144–151, ICST, Brussels, Belgium, 2011. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering).

[11] Maksims Fiosins, Jelena Fiosina, Jörg P. Müller, and Jana Görmer. Agent-Based Integrated Decision Making for Autonomous Vehicles in Urban Traffic. In Yves Demazeau, Michal Pechoucek, Juan Corchado, and Javier Perez, editors, *Advances on Practical Applications of Agents and Multiagent Systems*, volume 88 of *Advances in Intelligent and Soft Computing*, pages 173–178. Springer Berlin / Heidelberg, 2011.

[12] Alfredo Garro and Wilma Russo. easyABMS: A domain-expert oriented methodology for agent-based modeling and simulation. *Simulation Modelling Practice and Theory*, 18(10):1453–1467, 2010.

[13] Jan D. Gehrke and Christian Ober-Blöbaum. Multiagent-based Logistics Simulation with PlaSMA. In *Informatik 2007 - Informatik trifft Logistik, Band 1. Beiträge der 37. Jahrestagung der Gesellschaft für Informatik*, pages 416–419. Technologie-Zentrum Informatik, 2007.

[14] Amineh Ghorbani, Pieter W. G. Bots, Virginia Dignum, and Gerard P. J. Dijkema. MAIA: a Framework for Developing Agent-Based Social Simulations. *J. Artificial Societies and Social Simulation*, 16(2), 2013.

[15] Jana Görmer and Jörg P. Müller. Group Coordination for Agent-Oriented Urban Traffic Management. In Yves Demazeau et al., editor, *Advances on Practical Applications of Agents and Multi-Agent Systems (Proc. of PAAMS 2012)*, volume 155 of *Advances in Soft Computing*, pages 245–248. Springer-Verlag, 2012.

[16] Dirk Helbing, Ansgar Hennecke, Vladimir Shvetsov, and Martin Treiber. MASTER: macroscopic traffic simulation based on a gas-kinetic, non-local traffic model. *Transportation Research Part B: Methodological*, 35(2):183 – 211, 2001.

[17] Mark D. Hill and Michael R. Marty. Amdahl's Law in the Multicore Era. *Computer*, 41(7):33–38, July 2008.

[18] Michaela Huhn, Jörg P. Müller, Jana Görmer, Gianina Homoceanu, Nguyen-Thinh Le, Lukas Märtin, Christopher Mumme, Christian Schulz, Nils Pinkwart, and Christian Müller-Schloer. Autonomous agents in organized localities regulated by institutions. In *Digital Ecosystems and Technologies Conference (DEST), 2011 Proceedings of the 5th IEEE International Conference on Digital Ecosystems and Technologies*, pages 54–61, May 2011.

[19] Roberto Ierusalimschy. Programming with multiple paradigms in lua. *Functional and Constraint Logic Programming*, pages 1–12, 2010.

[20] Franziska Klügl and Paul Davidsson. AMASON: Abstract Meta-model for Agent-Based SimulatiON. In Matthias Klusch, Matthias Thimm, and

Marcin Paprzycki, editors, *Multiagent System Technologies*, volume 8076 of *Lecture Notes in Computer Science*, pages 101–114. Springer Berlin Heidelberg, 2013.

[21] Franziska Klügl and Frank Puppe. The Multi-Agent Simulation Environment SeSAm. In H. Kleine Büning, editor, *Proceedings of Simulation in Knowledge-based Systems*. Universität Paderborn, Reihe Informatik, Universität Paderborn, April 1998.

[22] Daniel Krajzewicz, Georg Hertkorn, C. Rössel, and P. Wagner. Sumo (simulation of urban mobility). In *Proc. of the 4th Middle East Symposium on Simulation and Modelling*, pages 183–187, 2002.

[23] Frederick Kuhl, Judith Dahmann, and Richard Weatherly. *Creating computer simulation systems: an introduction to the high level architecture*. Prentice Hall PTR Englewood Cliffs, Upper Saddle River, NJ, USA, 2000.

[24] Roberto Ierusalimschy Luiz Henrique de Figueiredo, Waldemar Celes. *Lua Programming Gems*. Roberto Ierusalimschy, 2008.

[25] Sean Luke, Claudio Cioffi-Revilla, Liviu Panait, and Keith Sullivan. Mason: A new multi-agent simulation toolkit. In *Proceedings of the 2004 SwarmFest Workshop*, volume 8, 2004.

[26] Markos Papageorgiou, Jean-Marc Blosseville, and Habib Hadj-Salem. Modelling and real-time control of traffic flow on the southern part of Boulevard Peripherique in Paris: Part I: Modelling. *Transportation Research Part A: General*, 24(5):345 – 359, 1990.

[27] Yoav Shoham. Agent-Oriented Programming. *Artificial Intelligence*, 60(1):51–92, 1993.

[28] V. S. Subrahmanian, Piero Bonatti, Jürgen Dix, Thomas Eiter, Sarit Kraus, Fatma Özcan, and Robert Ross. *Heterogenous Active Agents*. MIT Press, 2000.

[29] Seth Tisue and Uri Wilensky. NetLogo: A simple environment for modeling complexity. In *International Conference on Complex Systems*, pages 16–21, 2004.

[30] Gerhard Weiss, editor. *Multiagent systems*. MIT-Press, 2013.

[31] Michael J. Wooldridge. An Introduction to MultiAgent Systems, 2009.

A Appendix

A.1 Optimisation

Wooldridge describes in [31] some pitfalls in agent developing: (1) *"You forget that your are developing multithreaded software"*, (2) *"Your design does not exploit concurrency"*, (3) *"You have too few agents"*.

As discussed in Section3.2 there are two disjoint sets of objects in our simulation: non-synchronised and synchronised objects. Taking the above three statements seriously, our aim is to design a scalable, multi-threaded and multi-core system which can handle a large number of agents, that act concurrently.

With the technical restrictions (memory and number of threads), we need another approach, which is inspired by the technical view of the MMORPG in Section 3:

- We create a scheduler on its own to handle the agents. It is based on a thread-pool.
- We measure the average of the calculation time of each agent when it is active (counting the CPU cycles).
- Based on this result, we optimise the number of agents between the micro-kernels with a thread-/core-stealing algorithm (in future work we aim to describe this with a stochastic process).

After having defined one discrete simulation step, we denote this step "tick" and the process definition of one step is as follows:

```
wait for all kernels to be synchronised
do parallel: for each synchronised object
run object tick processing and determine CPU cycles

wait for all kernels to be  synchronised
do parallel: for each local non-synchronised object
run object tick processing and determine CPU cycles
steal request exists then send objects

while (other kernels are overload)
steal non-synchronised objects from other kernels
do parallel: for each stealed non-synchronised object
run object tick processing and determine CPU cycles
```

Each simulation object owns a (pre/post) tick method, which is called by the scheduler. There exist only two global blocking operations for synchronization over all kernel instances. Each micro-kernel process runs the (global)

synchronised objects first. After finishing the simulation environment is synchronised on each kernel. In the next step the kernel runs the non-synchronised objects. This second loop can be run in a fast way, e.g. the agents do nothing, so the micro-kernel idles, then the while-loop sends "steal requests" and gets objects from the other instances. Figure 9 shows the stealing process (bullets are the agents, with different calculation time).

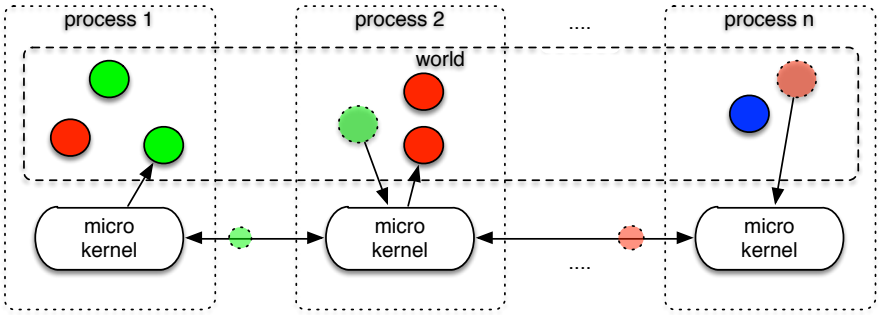


Figure 9: Optimisation

This idea allows the processing of a large number of agents with different (complex) plans and belief bases, because we can define the optimisation process with different targets and methods. The simulation consists of a finite number of discrete steps and objects, so we can describe the process with a discrete stochastic approach.

A.2 Lua implementation of the cow scenario

In this section we provide a first version of the Lua (pseudo) code for the AML meta-model as well as the *cow scenario* example. The source code can be found at [19]. As already mentioned we use a custom object-oriented structure with Lua. Of course, all classes on the micro-kernel level have a corresponding Lua object that we will however not explicitly specify here.

A.2.1 AML meta-model.

We will start with the `Entity` class from the meta-model. By convention our class names begin with a capital C, while private fields and methods begin with an underscore. We adopted this scheme for the sake of readability, since these are not natural concepts of Lua.

[19] `https://mecdev.rz-housing.tu-clausthal.de/gitlab/desim/maserati`

```
CEntity = class(CSynchObject)

CEntity.registered = {}
CEntity.actionToEffector = {}

function CEntity:_init(sensors, effectors)
  self._sensors = sensors
  self._effectors = effectors

  for k, effector in effectors do
    for act in effector:getAvailableActions() do
      self._actionToEffector[act] = effector
    end
  end
end

function CEntity:register(agent)
    if instanceof(agent, CAgent) then
        self.registered[agent.getUuid()] = true
    end
end

function CEntity:unregister(agent)
    if instanceof(agent, CAgent) then
        self.registered[agent.getUuid()] = nil
    end
end

--assumes a single effector per action
function CEntity:performAction(entity, name, parameters)
    local currentEffector = self.actionToEffector[name]
    if currentEffector ~= nil then
            currentEffector.performAction(name, parameters)
    end
end

function CEntity:getAvailableSensors()
  local sensorNames = {}
  for k, sensor in pairs(self._sensors) do
    table.insert(sensorNames, sensor.getName())
  end
  return sensorNames
end
```

```
function CEntity:getSensorData()
  local percepts = {}
  for k, sensor in pairs(self._sensors) do
    local p = sensor.getAllPercepts(self)
    for k, percept in pairs(p) do
      table.insert(percepts, percept)
    end
  end
  return percepts
end
```

We use a custom `class()` function to create a new class `CEntity` that is derived from the general class for synchronous objects. This is followed by a few functions which manage the entity's specific sensors and effectors. Agents can register with the entity and thus perceive and act upon the world.

```
CSensor = class(CSynchObject)

function CSensor:_init(name)
        self._name = name
end

function CSensor:getAllPercepts(entity)
  --entirely depends on sensor-type
end

function CSensor:getName()
        return self._name
end

--------

CEffector = class(CSynchObject)

function CEffector:performAction(entity, name, parameters)
        -- depends on specific effector
end

function CEffector:getAvailableActions()
  -- returns a list of strings
end
```

The classes for sensors and effectors are mostly skeletons only, since the precise functionality depends entirely on the particular scenario at hand.

```
CArtifact = class(CSynchObject)

function CArtifact:getManual()
  return self._manual
end

function CArtifact:use()
end

function CArtifact:_init(manual)
  self._manual = manual
end
```

The `Artifact` class is on a similarly abstract level only providing a skeleton that is to serve as a guide for deriving classes. It includes a manual that describes the functionality of the artifact which could be useful in cases where agents do not know the artifacts in advance.

```
CNorm = class(CSynchObject)

function CNorm:_init()
  self._active = true
end

function CNorm:isActive()
  return self._active
end

function CNorm:check()
  --check whether the norm triggers
end

--------

CAgent = class(CSynchObject)

function CAgent:_tick()
  self:_perceive()
  self:_think()
  self:_act()
end

function CAgent:_perceive() end
function CAgent:_think() end
function CAgent:_act() end
```

Norms and agents are also mostly dependent on the simulation developer. This allows for arbitrary agents (and norms) to be simulated. To illustrate this, we created a BDI agent that is derived from the general `Agent` class. Note that we left out some minor details.

```
CBDIAgent = class(CAgent)

function CBDIAgent:_init(entity)
  entity:register(self)
  self._entity = entity
end

CBDIAgent._beliefs = new(CBeliefBase)
CBDIAgent._events = {}
CBDIAgent._plans = new(CPlanBase)
CBDIAgent._intentions = {}

function CBDIAgent:_perceive()
  self_.latestPercepts = self._entity.getSensorData()
end

function CBDIAgent:_think()
  --update BB
  self.beliefs:_update(self._latestPercepts, self._events)

  if #self._events > 0 then
    --select event
    local event = table.remove(self._events, 1)
    local plan = self._plans:selectPlan(event)
    table.insert(self._intentions, plan:_toIntention());
  end

  while #self._intentions > 0 do
    --select intention
    local intention = table.remove(self._intentions, 1);
    local action = intention:getAction()

    if(self._act(action)) then
      break
    end
  end
end

function CBDIAgent:_act(action)
```

```
  if instanceof(action, CInternalAction) then
    action.execute()
    return false
  end
  if instanceof(action, CMessage) then
    CMessage.send()
    return false
  end
  --external action
  action.execute()
  return true
end
```

Finally, we show the abstract class `Area`.

```
CArea = class(CSynchObject)
CArea._parentArea = {}
CArea._subAreas = {}

-- private
function CArea:_init(parentArea)
  self._parentArea = parentArea
  self._publish("_parentArea",self.value)
end

-- public
function CArea:destructor()
  -- call destructors of all children
end

function CArea:newEntity()
  -- create entity
end

function CArea:getParentArea()
  return self._parentArea
end

function CArea:getSubAreas()
  return self._subAreas
end

function CArea:getNeighbours()
  return self.parent._subAreas
```

```
end

function CArea:createSubArea()
  -- create new child instance
  subarea = CArea:new(self)
  -- add child to list
  table.insert(self.getSubAreas, subarea)
end

function CArea:removeSubArea(subarea)
  -- do not deleted the first area
  if table.empty(subarea.getParentArea()) == false then
    table.removeEntry(subarea.getParentArea())
  end
end
```

This concludes the AML meta-model. We continue by using it to create a model of the cow scenario.

A.2.2 Scenario layer: Cow example.

We implemented switches and fences as artifacts while obstacles are implicit. The whole world and the corrals are each realised as an area.

```
Fence = class(Artefact)

function Fence:_init(area, x1, y1, x2, y2)
    Artefact:_init(self._fencemanual)
    self._area = area
    self._beginX = x1
    self._beginY = y1
    self._endX = x2
    self._endY = y2
end

function fence:keepOpen()
    self._open = true
end

function fence:_preTick()
    self._open = false
end

function fence:_postTick()
```

```
    if self._open == true then
        self.area.makePassable(self._beginX,
            self._beginY, self._endX, self._endY)
    else
        self.area.makeImpassable(self._beginX,
            self._beginY, self._endX, self._endY)
    end
end

--------

Switch = class(Artefact)
Switch._switchmanual = Manual:new()

function Switch:_init(area, fence, x, y)
    Artefact:_init(self._switchmanual)
    self._area = area
    self._fence = fence
    self._x = x
    self._y = y
end

function Switch:activate(activatingEntity)
    if self._area.getPosition(activatingEntity)
                == self.getPosition() then
        self._fence.keepOpen()
    end
end

function getPosition()
    return self._x, self._y
end
```

Furthermore we have one entity for each cow and cowboy respectively. In terms of functionality both are the same yet.

```
CowEntity = class(CEntity)

function CowEntity:new()
    local sensors = {new(ProximitySensor, self, self:getArea())}
    local effectors = {MoveEffector, self, self:getArea()}
    return self.super:new(sensors, effectors)
end
```

```
--------

CowboyEntity = class(CEntity)

function CowboyEntity:new()
    local sensors = {ProximitySensor, self, self:getArea()}
    local effectors = {MoveEffector, self, self:getArea()}
    return self.super:new(sensors, effectors)
end
```

For these entities a sensor that creates percepts based on what happens around the entity is sufficient. Also, the entities can only modify the world by altering their own position, thus a corresponding effector is implemented. Note that in our approach sensors are to *decide* what the current percepts of an entity are based on knowledge of the entire world. Likewise, effectors are responsible to decide whether the action they are to handle is feasible under the current circumstances.

```
MoveEffector = class(CEffector)

function MoveEffector:_init(entity, area)
    self._entity = entity
    self._area = area
end

function MoveEffector:performAction(name, parameters)
    local x = 0
    local y = 0
    if name == "move" then
        if parameters.direction ~= nil
                and type(parameters.direction) == "string" then
            local d = parameters.direction
            if d == "east" then
                x = 1
            elseif d == "west" then
                x = -1
            elseif d == "south" then
                y = 1
            elseif d == "north" then
                y = -1
            end
            if not(x==0 and y==0) then
                local entityX =
                    self._area.getPosition(self._entity).x + x
```

```
                local entityY =
                    self._area.getPosition(self._entity).y + y
                if self._area.isPassable(entityX, entityY) then
                    self._area.setPosition(entity, x, y)
                end
            end
        end
    end
end

--------

ProximitySensor = class(CSensor)

function ProximitySensor:_init(entity, area)
    self._entity = entity
    self._area = area
end

--visibility range
ProximitySensor.vRange = 5

function ProximitySensor:getAllPercepts()
    local entityX =
        self._area.getPosition(self._entity).x
    local entityY =
        self._area.getPosition(self._entity).y
    return self._area.getObjectsAround(x, y, vRange)
end
```

The `CowWorld` consists of an `area` with two subareas representing the two corrals:

```
CowWorld = class(CArea)

function CowWorld:_init(sizeX,sizeY)
  self.CowWorldGrid = DataType.Grid:new(sizeX,sizeY)
  self.CowWorldCorral1 = CowWorld.createSubArea("teamA",3,4,1,1)
  self.CowWorldCorral2 = CowWorld.createSubArea("teamB",3,4,sizeX-3, sizeX-4)
end

function CowWorld:createSubArea(label,sizeX,sizeY,startPosX, startPosY)
  self.label = label
  self.positionX = startPosX
```

```
  self.positionY = startPosY
  subArea = self.parent.createSubArea()
  subArea.CowWorldGrid = DataType.Grid:new(sizeX,sizeY)
end
```

Finally, a `Simulation` could look like this:

```
CowSimulation = class(CSimulation)

function CowSimulation:_init()
    cowWorld = CowWorld:new(25,25)
    fence = Fence:new(cowWorld, 10, 10, 10, 15)
    switch = Switch:new(cowWorld, fence, 10, 16)
    -- create cows
    -- create agents
    -- create entities
end

function CowSimulation:_tick()
  -- do something
end

function CowSimulation:_start()
  -- prepare the simulation start
end

function CowSimulation:_end()
  -- do something
end
```